AI on software engineering processes

Carles Miralles Pena

Advisor: Jordi Cabot

Tutor: Sergi Gomez

A thesis submitted for the degree of
Master of Science in Artificial Intelligence
January 2018

Factultat d'informàtica de Barcelona
Universitat Politècnica de Catalunya (UPC)

Factultat de matemàtiques i informàtica
Universitat de Barcelona (UB)

Escola tècnica superior d'enginyeria
Universitat Rovira i Virgili (URV)

Statment of Originality

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due references are made.

Abstract

To build software systems is a complex task in which many actors interact: customers, project managers, software engineers, designers, quality engineers, etc. To manage the construction of such systems exists different software methodologies, each one with its owns artifacts and philosophies. Waterfall model was the first kind of methodologies originated in manufacturer and construction industries. In waterfall, the software development cycle is divided in five differentiable phases or processes that are executed linearly one after the other: (1) requirements capture user needs (2) analysis models business rules and entities, (3) design obtains a software architecture, (4) coding implements the system, (5) testing ensures the correct functionality and (6) operations support the maintenance. The main problem of this methodology is that the interaction with client is minimum, the phases are too long and to incorporate changes that might appear along the project execution supposes a big deal. For that reason appeared another kind of more resilient methodologies, named agile, that tried to mitigate those inconveniences by means of putting much more emphasize in the change. Now, the same processes are shorter in time and the interaction with the client is frequent in order to get feedback from their expectations and adapt the system to fulfill them.

To control all these actors and processes is hard and often exists a correlation of forces among them. Typically each actor and/or process has its own language to tackle the same abstractions from different perspectives, thus might incorporate redundancies. Any redundancy adds complexity and are potentially dangerous since any change in it should be taken into account for the all references to it. In this thesis the focus of interest are two processes very related one each other: requirements and executable specifications (also known as tests).

Requirements are typically textual descriptions of the interactions between the user and the system, ie. what is the expected behavior from a software system when the user interacts with it. On the other hand, executable specifications are programs, ie. executable entities, that ensures that requirements in the system are fulfilled. Both have similar goals, ie. to specify software behaviour, however the language is different. While requirements are expressed in natural language, tests are written in some programming language. Thus, any change will affect both artifacts, and we saw that current software methodologies are meant to accept changes anytime.

In the past years appeared Gherkin, a new language that tries to put close both worlds. Gherkin introduces a domain specific language (DSL) to express requirements in a particular way: *Given preconditions When user interactions Then system expectations.* However, once a requirement has been written down, has to be manually translated into code, ie. the executable specification. This thesis explores the automation of this translation by means of a virtual assistant and the current technologies around it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to the large quantity of available information as well as the huge computational power achieved today, the application of Artificial Intelligence (AI) in computer programs has become, now more than ever, much popular. The overall goal of AI is to provide technology to build artificial systems, ie. machines and computers, that mimic natural intelligence.

Virtual assistants (VA) are one of these systems that nowadays is being adopted by most of the technological companies: SIRI (Apple), Alexa (Amazon) or any chatbot inhabitant in instant messaging (IM) systems as Slack or Telegram are some relevant examples. As a consequence, the number of platforms to build VA has increased in the recent years and many companies are releasing their own solutions to develop such agents on the top of other internet services.

The human-machine interaction is changing swiftly and we are observing an evolution in computer programs: from rigid applications that respond to limited user commands, to new ones capable to communicate in human language and with more autonomy to accomplish the tasks that they are meant to. However we are still in the early stages of VA usage and there are lot of room to apply this paradigm in many other the areas. Software engineering (SE) is the area of interest of this thesis and conversational bots the kind of VA chosen to tackle the challenges faced in it.

1.1 Motivation and Objectives

Software engineering is a field that defines and applies systematically methods to facilitate the construction and reasoning of large programs. It is a multi discipline that not only concerns coding but many other areas such as requirements acquisition, project management, testing, etc. The institute of electrical and electronics engineers (IEEE) defines in [17] fifteen different knowledge areas (KA) around SE. Among these KA we are particularly interested in two very connected: requirements and tests.

Requirements are artifacts that capture user needs and describe software systems behaviour and their interactions. On the other hand, software tests ensure the system behaviour, ie. *a program do what is meant to do*. While the former are often expressed in natural language along with a modeling language such as UML, the latter are written in a programming language that typically coincides with the one used to code in the main program. Therefore testing code can be seen as executable specifications.

This duality of the same problem results in redundancy that makes more difficult software maintenance. It might create uncertainty when there exists contradictions or incompleteness in one of two elements. Frequently as time passes and a computer program evolves, textual requirements become outdated and executable specifications the actual source of truth. Moreover, since both tasks are expressed in different languages, often are done by different roles in a company that ends up in doubling the number of required resources. In a context in which the demand of software engineers is higher than the human resources, any help to simplify the number tasks might have a high impact. Therefore, we believe that a tool to interpret requirements in natural language and translate them into the code automatically could improve the performance of a software engineer and reduce the risk of communication misunderstanding and thus program errors.

Finally, software engineers use extensively instant messaging systems as communication mechanism. They allow engineers to communicate both in real time when urgency is required and in deferred time when a task requires to avoid

distraction. Recently, chatbots have been reappeared as an important part of these IM systems to extend their functionalities and many companies are building chatbots in front of their services. Therefore, we are also convinced that human-chatbot interaction is a natural approach to implement the tool aforementioned.

This thesis has twofold objectives (i) to create a tool to translate textual requirements into executable specifications and (ii) design a virtual assistant to implement it.

Chapter 2

Background

In this chapter we outline the foundation of two main aspects in which this thesis is based on: **conversational systems** and **software requirements specification**. In the first part we introduce a brief introduction to agents and the particular case of virtual assistants. We also describe the main natural language techniques used by these assistants to conduct human conversations. In the second part we introduce the main ideas behind software requirements specification and their connection with software tests. In the last section of the chapter we review object oriented technology and three kind of specifications that this thesis is focused on.

2.1 Conversational systems

2.1.1 Agents: a brief introduction

In computer science the word software agents refers to programs that act on behalf of an user or another program. A closely related word is (software) intelligent agents or just agents, used to denote the presence of intelligence in such programs. Although there is no consensus in what an intelligent agent is, is quite accepted that autonomy is the central notion of the agency. However many programs act on behalf of an user and might have autonomy but not being considered them as intelligent agents, thus what are the differences between these two concepts?. Franklin wrote in [12] a detailed analysis around this question. We introduce here

a modest explanation based on both terms definition and two examples to show the main differences.

We define a **computer program** as a collection of instructions that perform different tasks when are executed by a computer. On the other hand an **intelligent agent** is a computer system situated in an environment, capable to sense it and act in autonomous way to meet their goals. We observe that programs are defined only by the tasks they bring about, however agents are specified by their environment, their sense capabilities, their actions and their goals. A very simple example of this is a spell corrector, that in its form of program is described as a software that corrects word mistakes when an user invokes it. Nevertheless the agent version might be described as software entity that is situated in one environment in which words are written in it, has capabilities to watch this environment, its goal is to keep the environment free of spelling errors, and its actions are to correct them whenever decides. Notice that both software entities have goals, since a task is in itself a goal, however regular programs do not possess autonomy to execute the required actions.

In the next example we introduce a scenario in which the program version presumes autonomy and might cause misconception. We assume two software entities that require to send a report to managers of a company. In one version (i) the report should be sent every Monday while in other (ii) requires the report only under anomalous conditions. We observe that two examples are executed autonomously without explicit user intervention. Moreover both share the same goal and environment, ie. to send a report to managers in a company. However in (i) the rule condition 'every Monday' triggers the action of sending the report regardless the company in which is situated. On the other hand in (ii) the rule condition 'anomalous conditions' is not so precise as in the previous case and will depend on term definition given by the company. Thus in (i) a simple list of rules would be enough to implement the program activation for a given set of events, while in (ii) other techniques to model the environment conditions that turn out in the fuzzy term 'anomalous conditions' are required. So that, the fact of executing without human intervention does not imply autonomy but the way

in which responses to changes in the environment is what determines this degree of freedom.

Typology

There exists multiple ways of classifying agents, we introduce in this section the classification proposed by Nyacinth S Nwana in [15]. The author defined five dimensions to classify agents: mobility, role, attributes, reactivity and hybrids.

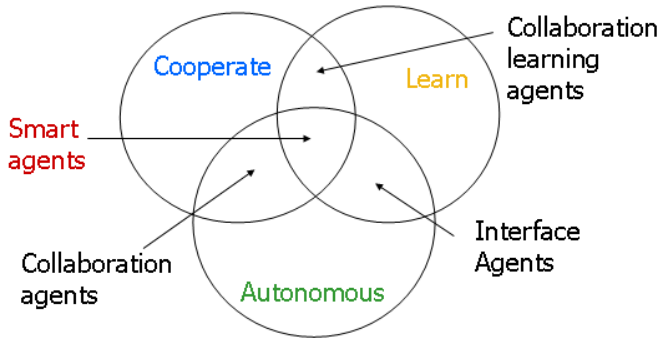
The first attribute, mobility, lead us to classify agents in **mobile** when they can move through a wide area network or contrary as **static**. Role makes classification according to the agent goal, eg. information agents characterised by managing big amount of data obtained from internet. In terms of agent attributes Nwana considered three attributes in its classification, **autonomy, cooperative and learning** that their combination results into three[1] main kind of agents (fig. 2.1a):

- Collaborative agents: these kind of agent form groups to accomplish more complex tasks than they can solve by their own. They require autonomy and communicative skills to negotiate with other agents.

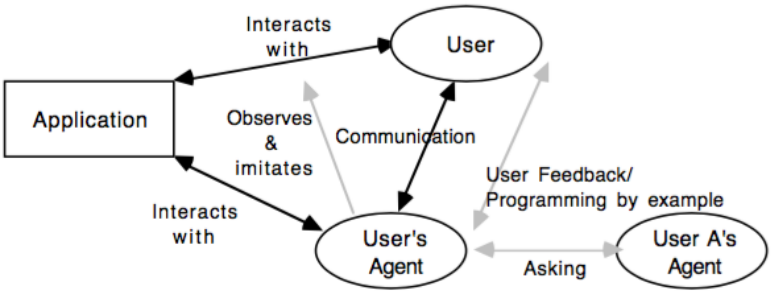
- Interface agents (IA): also known as personal or virtual assistants (VA), are meant to help users to solve different tasks. They should learn from the user feedback and be proactive to offer their services when they are useful. In fig. 2.1b is depicted the main communicative aspects of an IA. They might observe user interaction with other applications and communicate to user certain situations, eg. a VA learns from the user interaction with a traffic application and warns in the future when the user is going to a crowded area. Finally IA might communicate with other agents.

- Smart agents: they would have to learn how they react or interact with the environment.

[1]He discarded collaborative learning agent arguing that the absence of autonomy makes impossible to imagine the other two attributes



(a) Agent typology based on attributes



(b) IA interactions

Figure 2.1: Aspects of interface agents (IA) [15]

The fourth dimension distinguishes between reactive and deliberative agents. While the behaviour of the former type is driven by a simple list of action-reaction rules, the latter presents a complex decision subsystem. In the following section we introduce the kind of architecture required to implement each type. The last category is referred by the author as hybrid when multiples philosophies are combined, eg. We might want a static reactive internet interface agent for a virtual assistant that search internet information without movement through the network and following simple reaction rules.

Internal architecture

The notion of internal architecture refers how the action rules that drive the agent response and the internal state is hold. We have seen that agents might be classified as reactive or deliberative that is a critical aspect for the architecture. We introduce here the classical vision of this concept proposed by Russel and Norving in [16] in which the authors defined five kind of architectures, the first

two to support reactive agents and the rest for deliberative agents:

Simple reflex agents: Agents only act basis on its current percept of the environment, ignoring previous history. The agent function is based on a set of condition-action rule.

Model-based reflex agents: Agents keep an internal representation of the environment that describes the part of the world that cannot be seen. This model allows the agent to handle partially observable environments. The next action to execute will depend on the current perception and this internal state.

Goal-based agents: These agents expands model-based architecture adding support to manage goal information. Goal information describe desirable situations, so this architecture allows the agent choose the action that reach a goal when multiple options are given.

Utility agents: If the previous architecture keep a binary classification between goals and non-goals, this architecture models different degree of desirable situation. This is able by using utility functions which maps any state to be selected to a measure of utility, ie. how happy is would make the agent the selection of a given state. Notice that goal-based selection is crisp while this is fuzzy.

Learning agents: This architecture is the most complex and allows the agent to start in unknown environments and to evolve its competence to make decisions. Is form by four blocks: critic, learning, performance and problem generator. Critic responsible to control how the agent is doing based on an standard performance. Learning is the block that makes the agent evolve and improve in the time. Performance is the block that decides which is the next action to be taken according and problem generator suggest actions that will lead to goals.

2.1.2 Chatbots

ELIZA is with no doubt the most well-known chatterbot related to the AI field. The bot was created by Weizenbaum [18] at the MIT and simulated a psychotherapist. The chatterbot imitated conversations by means of pattern matching and substitution methodology, ie. scripts that answer adding words extracted from previous user inputs. It was one of the first programs capable of passing the Turing Test and counts with a large number of implementations [2].

Chatterbots are a kind of virtual assistant and its main goal is to assist an user by means of textual conversations. During the course of a conversation the chatterbot should be able to understand user inputs and answer according its goals in a human way. Such conversations are possible because the use **natural language processing** (NLP) techniques.

Natural language understanding

Natural language understanding is a subtopic of NLP that deals with reading comprehension. Many techniques have appeared during course of the history, from regular expressions to filter keywords in a text to more advanced approaches based on statistical word occurrence.

Due to huge quantity of textual that we have available today, statistical approach is the main technique for NLU. In the context of chattebots there are two NLP tasks always present in these system to conduct dialogues:**intent classification** and **entity recognition**.

Intent classification

Intent classification is a NLP task that consists in classifying text according the main idea or topic expressed in a sentence. For instance, in a question answering (QA) system may be wanted to classify question in different areas of interest, eg. maths, computer science, physics, arts and literature .

Entity recognition

Entity recognition is the another NLP task that allows conversational agents to identify significant parts in a sentence for context understanding, ie. detection of semantic categories for a given text. In our QA system an user might wanted to ask a question about history, thus the name of people and dates would be important to give an automatic answer. Name entity recognisers (NER) provide a set of entities previously trained that they are able to identify, eg. PERSON, ORG, DATE, MONEY for people, organisations, dates and money respectively. However the list might be limited and insufficient for other contexts requiring to define new ones.

One part of this thesis consists in (i) to select relevant examples for training the system and (ii) to identify and label entities required to produce code automatically.

2.2 Software requirements specifications

A software requirement specification (SRS) is a description of the computer program to be developed. Typically contains a set of use cases describing the interaction between an user with the system that should be useful to guide the program development. It might be a starting point for software engineers developing the application, for designers prototyping the user interface or for customers and project managers to facilitate reviews.

There are two main ways of writing specifications: **formal** and **informal**. The former are expressed in a mathematical language as first order logic that provide a powerful tool to proof software correctness. However this way is complex to write, often is easier to write the program than the formal specification, and difficult to read for non technician. On the other hand, informal SRS are expressed in natural language with some additional diagrams written in any modeling language such as Unified Modeling Language (UML). Informal specifications are easier to write and read, however they can present ambiguities due to the use

of natural language. While formal specification might be used in critical systems such as a rocket software control, many companies that require agility and short time developments write specification using the second approach.

The focus of this work is semiformal specifications proposing a tool to analyse them and translate into executable specifications.

2.2.1 Executable specifications

Executable specifications, also named tests, are a set of executables that ensures program requirements, ie. the program does what is expected. **Test-driven developemnt (TDD)** is the main methodology that encourages the use of tests, ie. test-first programming. In fig. 2.2 is depicted the classical flow of this methodology:

1. The software engineer writes a test validating a requirement from the SRS. Since the main code is non existent the test fails (red).
2. The software engineer writes the minimum code to pass the test (green).
3. The code might be improved in terms of performance, readability or whatever important to be aligned with the rest of the code.

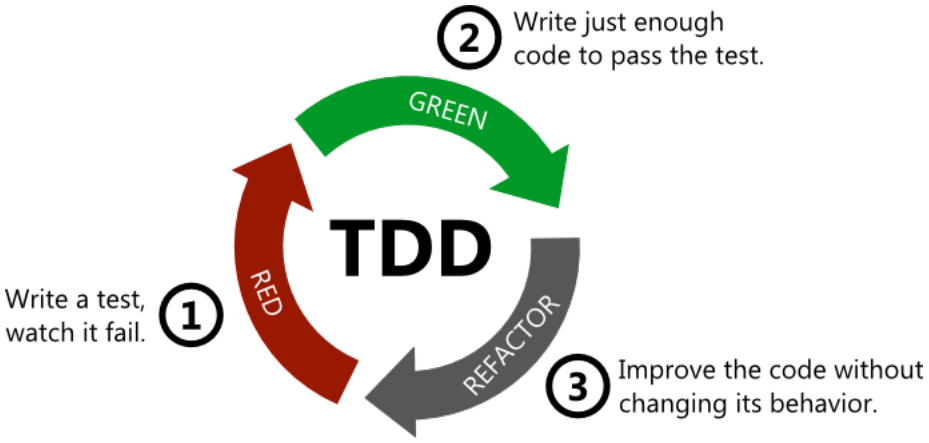


Figure 2.2: Test-driven development flow [10]

,

Although this technique of writing software is quite old, Kent Beck, the creator of extreme programming, was who rediscovered it in [7]. Nowadays all soft-

ware frameworks incorporate a sub system dedicated to write executable specifications since are considered an important part of the software engineering processes. Moreover some universities use this approach to teach students to program: in [11] is concluded that the number of defects in proposed exercises is reduced up to 45% and in [14] is stated that teaching TDD does not requires an additional time effort and better software models are observed in students homeworks.

The main advantage of executable specifications is that the principal software is protected from future changes. For instance, imagine a program that implements a social network in which people have contacts and messages are only allowed between linked people. One specification may wanted to test that two people that are not connected can not send/receive messages. If someone accidentally would change the main code obtaining the opposite behaviour, that is let messaging between unknown people, the previous specification will fail warning that this contract is not being satisfied.

One additional advantage is that executable specifications might be the actual source of truth of a software system. Textual requirements are typically written by a project manager and then a software engineer translates such requirements into executable specification. This duality requires of synchronisation that not always happens. For example, we might be in a situation during the development phase in which is realised that a requirement is too complex in technical terms or by mistake contradicts a business rule. The team decide to change the original requirement, ie. simplify it or rewrite it to remove the contradiction. Since these decisions are made during development phase is very common to incorporate such changes only in the executable specification, forgetting the original requirement. As a result of this lack of synchronisation, requirements might suffer a kind of degradation in time and tests become the actual specification. However such specifications are coded in a programming language and refers to internal entities that only technical people can understand. Languages as cucumber, based in Gherkin notation, tries to put together textual requirements and executable specifications. Nevertheless such language still requires a manual translation from textual requirement to specification.

2.2.2 Testing levels

We identify in [6] five levels of testing each one meant for validating different aspects of a program: Unit testing, Integration testing, Component interface testing, System testing and operational acceptance testing.

Unit testing is meant to validate specific units of code as functions or classes in object-oriented environment (OO). A test validating methods of the class stack would conform the unit testing for such class.

Integration testing seeks to validate interfaces between multiple software components. A test validating a payment gateway with the e-commerce system would be an example of this kind of testing.

Component interface testing are focused on test a component interface against different type of data that the component might get. Typically a database with real data and edge cases is hold for this kind of tests.

System testing are interested in validate that the system meet its requirements.

Operational acceptance testing validates the readiness of a product that is going to be released. Things as installation and backout are checked with these tests.

Since unit and system testing are the most extended tests, and the resources able to bring about this thesis are limited, the present work only covers these two types, leaving other levels as future work.

2.2.3 An example

In the following section we want to introduce an example of a textual requirement and its equivalent executable specification.

Gherkin example

The example shows how to describe some properties of a stack in Gherkin notation [3]. Properties are expressed in a quasi-natural language using some few key words to denote the intention of the sentence: **Given** to express a state that should be satisfied beforehand, ie. preconditions, **When** to specify some action with the system, **Then** to describe the expected behavior, ie. postconditions. The language also uses **And**/**but** to connect several sentences of the previous types.

```
Specification: Stack
Given a new stack
When we call empty method
Then true is returned

Given a stack
When an element e is added
And we call top method
Then e should be returned

Given a stack with N elements
And element E is added
When pop operation is called
Then is expected to returns E
And the new size of the stack is N-1
```

RSpec example

In this example we show the executable specification of the previous requirement written in RSpec [9]. RSpec is a domain specific language (DSL) implemented in Ruby programming language. However there exists many other DSL in other programming languages to code executable specifications: JUnit and JBehave (Java), chai and mocha[2] (Javascript), etc.

```
description Stack {
    # first given/when/then
```

[2]The bot code implemented in this work has been tested with this DSL

```
it {
    stack = Stack.new
    expect(stack.empty?) to_be true
}

# second given/when/then
it {
    stack = Stack.new
    expect{stack.add(1).top} eq 1
}

# third given/when/then
it {
    stack = Stack.new
    stack.add(E)
    l = stack.len

    expect{stack.top} eq E
    expect{stack.len} eq l-1
}
```

2.3 Model and services

To finalize this chapter we introduce the basic concepts in object oriented technology that this thesis is based on.

Models are software entities, ie. classes in object oriented technology, that represents objects from real life. They contain attributes to hold the internal state and methods to modify or to consult such state. In figure 2.3 is depicted two examples of this concept.

Services are also classes to implement all the functionalities in a software system, for example: In figure 2.4 a CreateBankAccount implements a functionality which receives a person and an initial balance, and is responsible for creating a new instance of model bank-account and associating it with the person. Services typically refer multiple models and can call different methods of that models to implement the whole functionality. In the he bibliography this way of implementing a whole functionality or algorithm within a class is named strategy pattern [13].

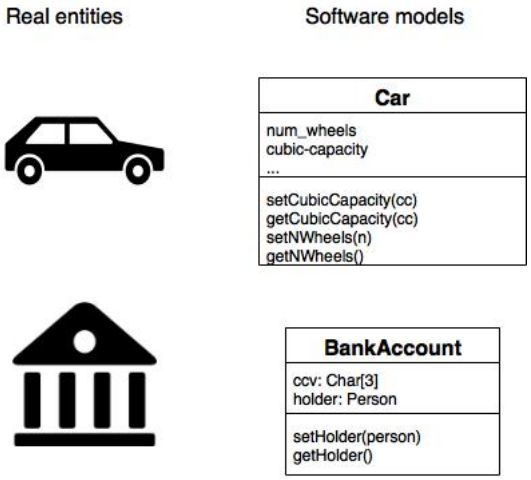


Figure 2.3: Software models mapping entities from real world

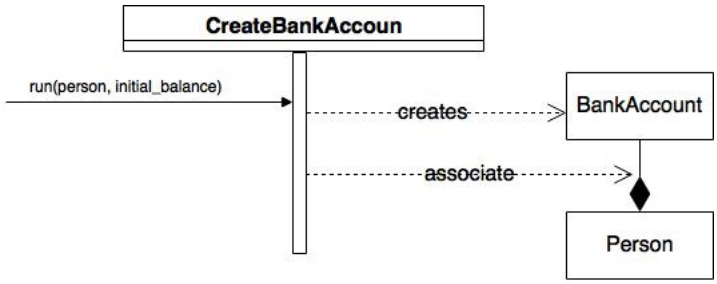


Figure 2.4: CreateBankAccount service

2.3.1 Models and services specifications

In this section we introduce three kind of executable specifications used to specify software systems: **model validations, methods and service specs**[3].

Model validations are meant to specify which properties should satisfy data values of a model to be valid in our system. There exists many kind of validations such as, (i) duplicates, eg. two users with the same identification number are not allowed; (ii) required attributes, eg. nulls in attributes id, firstname and lastname in a user are not allowed, or (iii) format validations, eg. a email in a user should be as "xxx@yyy.com". In this thesis only required attributes are taken into account and for that, the term model validations is misused to refer it.

Methods specifications are meant to describe the expected functionality of a method in a model. For example a specification for the method getCubicCa-

[3]sometimes authors refer them as controller specifications

pacity() might validate that the current value of the attribute cubic-capacity is returned.

Finally service specifications describe the expected behavior of some functionality in the system, eg. We expect that after calling the system functionality createBankAccount there will exists a new bank account in the system, associated with the person that was passed.

In the next chapter we will see examples in Gherkin of these three types of speceifciations.

Chapter 3

SpecBot: System Specification

In the previous chapter we have reviewed the bases of this thesis: conversational systems and executable specifications. In this chapter we introduce the requirements of our system as well as a bot architecture.

3.1 Requirements

We want to build a chatbot capable to capture textual requirements in Gherkin format and translate them into executable specifications in RSpec. We name our virtual assistant as Specbot.

Specbot has to conduct a conversation in natural language, understand user sentences and get those elements that are required to build the executable specification. We identify two user profiles depending on the level of expertise they have ie. junior and expert users. The former has no prior knowledge to write textual requirements in the instant message system. The latter has experience describing textual requirements in Gherkin notation, thus does not require any bot guidance. Any missing information required to generate the executable specification should be asked for by Specbot. When a requirement has been completed, Specbot should provide its equivalence in RSpec. Specbot will be trained for a limited kind of sentences.

We are focus on Object Oriented (OO) technology in which code is organised

through classes. Specbot should distinguish textual requirements for models and services. A model is a class describing a real world entity such as Cars, Houses, Bank accounts, etc. Services classes implement system functionalities such as bank account withdrawals, new users in the system, etc.

We identify three kind of specifications: model validations, model methods and services. Model validations let users describe what are the attributes of a model that are required to get a valid model. In the following example we want to ensure that all the users in the system will have firstname, lastname and identification-number:

validation spec

```
Given an user with firstname, lastname and identification-number
Then is expected to be valid.
```

Method specifications describe the expected functionality of one method within a model. In the following example the specification checks the expected behavior of the operation sum in a model calculator.

method spec

```
Given a calculator
When the method sum is called with parameters 2 and 3
Then the result should be 5
```

Service specifications describe system functionalities and might concern multiple models. The example checks the expected behaviour of the functionality get_users when two users are linked each other in the system.

service spec

```
Given an user u1
And an user u2
When u1 is linked to u2,
Then the functionality get_users with u1 includes u2
And get_users with u2 includes u1
```

Each specification has one or more model instances, ie. *clause Given*, and one or more expectations, ie. *clause Then*. Model instances might have many attributes and might be initialized to define an initial state in a particular scenario of a requirement, eg. *given a number with internal state five the operation next returns six*. Model expectations are meant to check expected values of a symbol, in the previous example *the value six*. A symbol might be an attribute or a method in a class instance for checking expectations from model state or method results respectively. In figure 3.1 is depicted the UML class diagram.

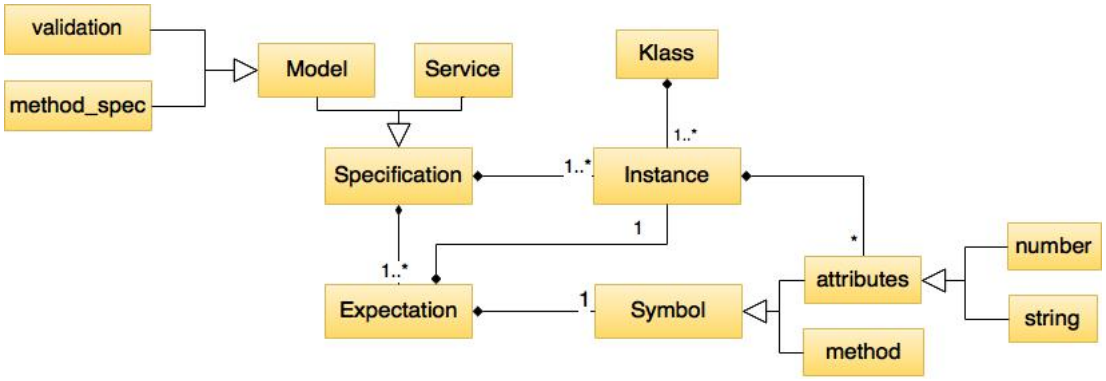


Figure 3.1: SpecBot classes diagram

3.2 Bot architecture

In this section we propose an architecture to implement Specbot. We have identified four distinct blocks: Instant message application, bot logic, NLP unit and context management unit (fig. 3.2). In the following sections we describe their functionality in detail.

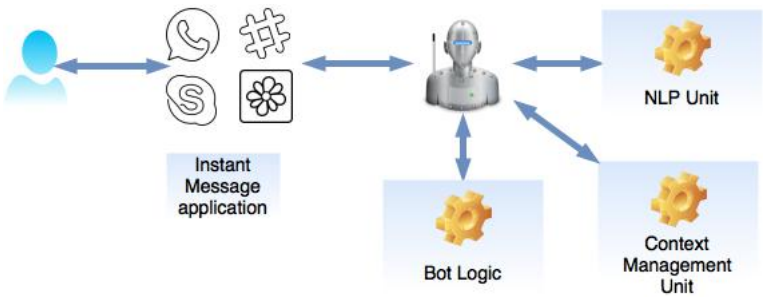


Figure 3.2: Chatbot architecture

3.2.1 Instant messaging application

Instant message application is the environment in which a chatbot lives. In such environment a chatbot can receive messages from an user and send to them. A message is a sentence in natural language, however the majority of IM platforms allow to send files, forms such as a question with two button answers or more sophisticated elements. In state of the art section we explain more details about this part.

3.2.2 NLP unit

In the section 2.1.2 we have seen that two are the main NLP tasks used by chatterbots to conduct conversations: intent classification and entity recognition. In the context of this thesis, these tasks are translated in the NLP unit as: (i) to identify what kind of specification is being described, (ii) to recognise relevant parts of each sentence needed to write the executable specification. Particularly, given an user utterance the NLP unit has to classify into three kind of specifications: **model validation**, **model method** and **service test**. Moreover, for each user sentence should be able to identify entities in it such as class names, attributes, etc. In the chapter 4 a detailed explanation of each entity is given.

In figure 3.3 is depicted the NLP unit including both main tasks: (i) the system classifies in three categories and (ii) identifies different entities (class name, method_name,..).

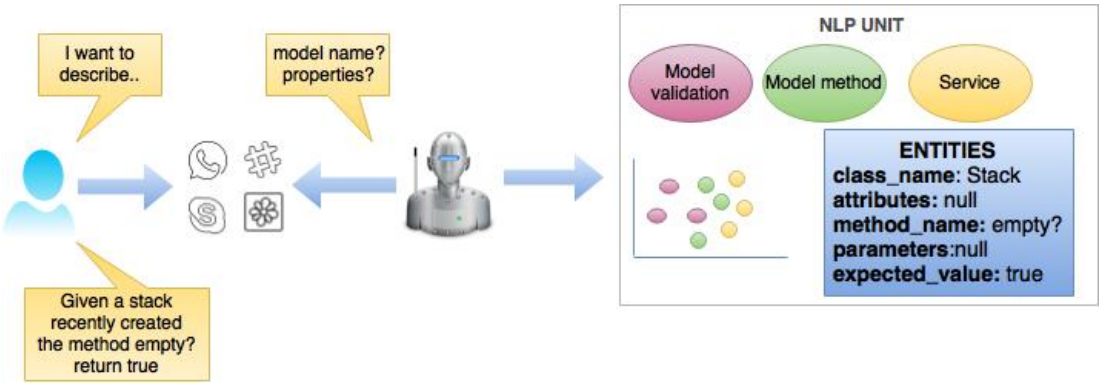


Figure 3.3: NLP Unit

3.2.3 Bot logic unit

The main responsibilities of the bot logic unit (BLU) are to implement bot conversations as well as bot responses to system events.

Conversations are written in a script language that uses a template system to specify bot sentences, user inputs and scripts flows. In the context of Specbot, four scripts have been defined: **select specification**, **refine validation**, **refine method** and **refine service**. Select specification defines bot texts to offer help and request one software requirement. Refine validation and method are for training the user to use Specbot to write these two kind of requirements, thus interacts with the user to get the entities required in each type. Refine service just asks for a textual requirement in Gherkin and shows the executable specification. Is worth mentioning that conversational scripts can also define a logic flow in each bot-human interaction, ie. decide where to jump for a given textual input. However we have situated this logic into the context management unit since this block is responsible for this task, ie decide next step. Moreover a script language presents some limitations explained in section 4.3.

In the BLU we should also define how a conversational script is triggered. In botkit each conversational script has a list of words and regular expressions that triggers the script execution when the user enters a word matching any element in the list, eg. {[Bb]ye, Sayonara, [Gg]ood[Bb]ye} $\Rightarrow$ exit script.

Finally here we might implement any functionality in response to any bot event, eg. before/after execute an script, etc. Specbot only reacts to one entry point which is a salutation word, the rest of the dialogue is controlled by the context management unit.

3.2.4 Context management unit

The last element in Specbot's architecture is the context management unit (CMU) that its main goal is to decide the next bot action based on previous decisions, ie. an internal state. We identify three responsibilities: (i) to control the flow

for each user-bot dialogues, (ii) to keep the internal state needed to generate the specification and (iii) translate them into executable code.

In the Specbot's context three flows have been defined: **select specification**, **refine validations** and **refine method**. Select specification gets the user input and pass it to NLP unit to figure out which kind of specification is being described and which entities are given by the user. Refine validation flow deals with validation specifications, eg. *Given an user with firstname, lastname and identification-number Then is expected to be valid*. Its goal is to obtain from the user all the required entities that has not been facilitated by the user or captured by the NLP unit. In the example Specbot should obtain examples of values for each attribute: firstname, lastname and identification-number. Finally, refine method controls the flow to get all the required entities in model method specifications.

To keep the internal state memory is required and to translate textual requirements into executable specifications a partial number of classes described in figure 3.1 have been implemented. The details of this part will be described in the following section.

3.2.5 Units dependencies

So far we have seen all the elements in Specbot's architecture, this section describes how the blocks are inter-related one each other.

The bot logic unit is the entry point of the bot and contains all the conversational scripts as well as the event handlers. The BL unit pass the control to the context management unit that should be decide what is the next step in the flow to be executed. However CMU might require analyse textual parts of the utterance so that might use the NLP.

In figure 3.4 is depicted the three blocks, their functionalities and their uses dependencies.

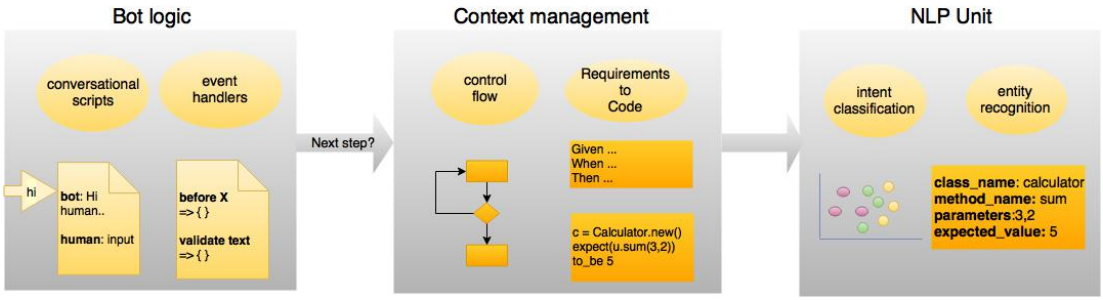


Figure 3.4: Bot components

3.3 State of the art

3.3.1 Botkit

There exists many companies offering IM applications with capabilities to develop and integrate bots such as Slack[1], Facebook messenger[2] or Telegram[3]. Since bots are a way to extend IM functionalities by third parties and every company pursues to have the maximum number of clients, the creation of bots has become a very important aspect nowadays. Therefore every IM company provides a proprietary application programming interface (API) to let programmers create and deploy bots in their systems.

The diversity of API makes difficult the programmer life's who, as in any other contexts, has to decide their preferences and learn one or more APIs. Therefore is preferable to use a middleware ie. a software to abstract the programmer from each IM details instead of multiple APIs (fig. 3.5). We have used botkit[4] as middleware for the prototype developed in this thesis because comes is an open source project used by 10.000 bots and has plugins for multiple NLP systems.

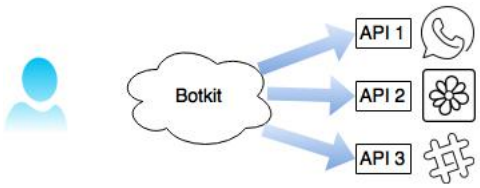


Figure 3.5: Botkit middleware

[1] https://slack.com
[2] https://www.messenger.com
[3] https://telegram.org
[4] https://botkit.ai

3.3.2 Rasa/NLU

With the popularization of virtual assistants which requires human language understanding, the number of tools to process textual input has been incremented. Nowadays the main tech companies have released their own solutions as a service: Microsoft LUIS[5], Amazon Lex[6], Google Dialogflow[7], IBM watson[8] and Facebook wit[9]. All these tools offer similar functionalities already introduced in section 2.1.2, ie. intent classification and entity recognition. In the current work we have used an open source library, Rasa/NLU, that offers the same functionalities built on the top of two other libraries: MITIE and Spacy. Rasa library offers more control over the NLP pipeline, ie. what subtasks are executed to recognise entities or classify utterances [5]. Moreover, since is a programming language library, programmer might decide whether should be installed locally or in a server to offer a cloud service as the other competitors.

3.3.3 Classifiers for NLP

As we have seen the NLP unit makes two kind of classification: sentence and named entities. We found in the literature the application of the classical ML algorithms for solving both tasks. Zhang et al. presents in [19] a study to classify question in 6 categories using five algorithms: Nearest neighbors, Naive Bayes, Decision Tree, Sparse Network of Windows and Support Vector Machines. Carreras et al. presents in [8] a named entity extractor using Adabost algorithm.

Rasa/NLU uses different libraries, ie. Mitie and Spacy, and algorithms depending on the task to be solved. For intent classification both libraries use a multiclass SVM with a linear kernel. However for named entity classification, Spacy uses conditional random fields and Mitie uses a SVM. This thesis only works with Spacy because is faster when number of examples is large.

[5] `https://www.luis.ai`
[6] `https://aws.amazon.com/documentation/lex/`
[7] `https://cloud.google.com/dialogflow-enterprise/`
[8] `https://www.ibm.com/watson/services/conversation/`
[9] `https://wit.ai`

Chapter 4

System Implementation

In the previous chapter we have seen *what* are the requirements of Specbot and *what* is its architecture. Now is time to describe *how* each unit has been implemented.

4.1 Introduction

In section 3.2 we have describe the logical blocks that conforms Specbot, here we will see how they have been implemented. The whole system is formed by three internet services: an IM application, the bot and the NLP unit. They use hyper text transfer protocol to send and receive messages between them, thus can be installed distributed in different machines. The bot send and receives messages to and from the user through an external instant messaging system. Although botkit abstracts us from IM platforms, we have only tested with Slack client. In figure 4.1 is depicted the physical systems.

4.2 NLP unit

In section 3.2.2 we viewed that this unit has two goals: (i) identify the kind of specification wanted by the user and (ii) identify parts required to build the specification from the textual input. Such tasks have been done using two NLP techniques: **intent classification** and **entity identification**. Since both tasks

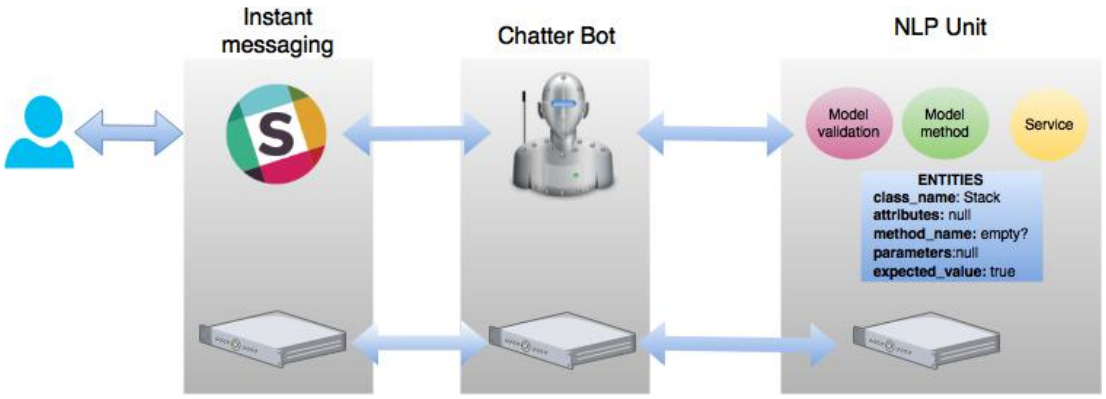


Figure 4.1: Physical systems

are approached by supervised learning algorithms, we need to provide a ground labeled data to train the system. An important part of this thesis has been to identify and label the required entities in the text. In the following sections is described which kind of sentences and which labels have been used to train the NLP unit. We propose for each kind of requirement a limited set of sentence structures and a set of custom entities to train the recogniser. Although the set is limited, many requirements might be expressed in Gherkin.

4.2.1 Model validation requirements

These requirements are meant to specify which are the attributes in a model that are required to get a valid instance. The following example shows a requirement that ensures that all the instances of the model computers will have values in attributes CPU, RAM-capacity, HD-capacity,screen-resolution and price:

validation in Gherkin

```
Given a computer
when I create a new instance with CPU, RAM-capacity, HD-capacity,
screen-resolution and price
Then is expected to be valid.
```

And the associated executable specification in RSpec:

validation in RSpec

```
it {
c = Computer.new(CPU: 'i7', RAM-capacity: '16Gb',
```

```
HD-capacity: '128Gb', screen-resolution: '1080dpi', price: '1000$')
expect(c.valid()) to_be true }
```

Notice that in the Gherkin requirement, the values of the attributes are not included, however they are mandatory to instantiate a class with attribute names in any programming language. Therefore the NLP unit has also been trained to understand specifications with attribute values such as:

validation spec

```
Given a computer
when I create a new instance with CPU: 'i7', RAM-capacity: '16Gb',
HD-capacity: '256Gb', screen-resolution: '1080' and price: '1000$'
Then is expected to be valid.
```

Custom entities

To transform any textual requirement into executable specification in Rspec, we identify three kind of entities to be labeled: **model name** (computer in the example), **attribute names** (CPU, RAM-capacity, HD-capacity, screen-resolution and price) and **values** ('i7', '16Gb', '256Gb', '1080' and '1000$'). In the next example is shown a tagged example in Gherkin:

Entities in validation models

```
Given a [project](model_name)
When is created with attributes [name](att_name): ['p1'](att_value),
[num-of-tasks](att_name): [2](att_value)
Then should be vallid
```

Structure of sentences

We have built a dataset composed of 54 utterances that follows the structure specified in table 4.1. The dataset contains examples up to a maximum of four attributes and only simple data values (*val*) have been taking into account, ie.

Structure	Comments
Given a MN When I $VB1$ with ATT_1 [,ATT_i] and ATT_n Then EXP	Attributes without values
Given a MN When I $VB1$ with $ATT_1 : val_1$ [,$ATT_i : val_i$] and $ATT_n : val_n$ Then EXP	Attributes with values
Given a MN When $VB2$ with ATT_1 [,ATT_i] and ATT_n Then EXP	Attributes without values. Passive
Given a MN When $VB2$ with $ATT_1 : val_1$ [,$ATT_i : val_i$] and $ATT_n : val_n$ Then EXP	Attributes with values. Passive
I want to describe a validation for MN model	Utterance expressing an intention not a requirement.

Table 4.1: Structure of validation specs (square brackets to express optionality)

Abbreviation	Examples
MN	car, bank-account,...
VB1	create, instantiate, ...
EXP	I expect to be valid, should be valid
VB2	is created, is instantiated, is initialized
ATT	age, num-tasks
VAL	3, 'John'

Table 4.2: Data examples

strings and numbers. We have also included utterances to expressg an user intention as in the last example in 4.1. In this situations, Spebot will ask to the user all the missing entities directly to the user, ie. the model name, the attributes and the value.

4.2.2 Method model requirements

This kind of requirements describes the expected behaviour of a method within a model. The following examples show a specification of a method get-extension within the model user using Gherkin notation and the translation into RSpec.

method requirement in Gherkin

```
Given an user
When I call the method get-extension with parameter '+34 66663616'
Then should return '+34'
```

method specification in RSpec

```
it {
  u = User.new()
  expect(u.get-extension('+34 666 636616')) to_be '+34'
}
```

Custom entities

To transform textual requirements into executable specification we identify four entities: user as **model name**, '(034) 666 636 616' as **parameter**, get-extension as **method name** and '(034)' as **expected result**. In the following example in Gherkin is showed a tagged requirement with these entities.

Example of method labeling

```
Given an [user](model_name)
When the method [get-extension](method_name) with parameter
['(034) 666 636 616'](par_value) is called
Then should return ['034'](expected_value)
```

Structure of sentences

In the table 4.3 is showed the structure of the sentences used to train this kind of specifications. A total of 56 utterances has been included in the dataset.

4.2.3 Service requirements

The last kind of specification are service tests which describes the expected behaviour of a functionality in the software system. It differs from the previous requirements because might contain references to several models by means of the connective *And* in the *Given* clause. Moreover, the user might call different

Structure	Comments
I want to VB a method ME [to MN]	Example expressing intention
Given a MN When I call [the method] ME with [PAR] PV_1 [,PV_i] and PV_n Then EXP	Active voice
Given a MN When [the method] ME is called with [PAR] PV_1 [,PV_i] and PV_n Then EXP	Passive voice

Table 4.3: Structure of utterances in methods specs

Abbreviation	Examples
VB	describe, add...
MN	car, bank-account,...
ME	get-extension, getname,...
PAR	parameters | params
PV_i	3, 'John'
EXP	should return '34' | 'Maria' is expected

Table 4.4: Data examples

methods of different instances in the clause *When*, so that has to specify which instance is referring each method. The next example is a requirement in Gherkin that ensures that the service get-cars returns the cars from one person, followed by its equivalent in RSpec.

Example of service in Gherkin

```
Given a person p1 And a car c1
When c1 is associated with p1 And I call get-cars with p1
Then c1 should be included in the result
```

service specification in RSpec

```
it {
  p1 = Peson.new()
  c1 = Car.new()
  p1.associate(c1)
  expect(GetCars.run(p1)) to_include c1
}
```

Custom entities

To translate from Gherkin to Botspec we identify the next entities in the next tagged example: In *Given* **model-name** (person and car) and **instance references** (p1 and c1). In *When* **constructor** (is associated), **parameter** (c1), **instance** (p1), **service** (get-cars) and **parameter** (p1). In *Then* **expected-value** (c1) and **method** (included). Notice that if the constructor is given in active voice, eg. "p1 associates with c1", then the parameters are interpreted in the opposite (p1 as instance and c1 as parameter).

Example of service labeling

```
Given a [person](model-name) [p1](instance)
And a [car](model) [c1](instance)
When [c1](parameter) [is associated](constructor) to [i1](instance)
And I call [get-cars](service) with [p1](parameter)
Then [c1](expected-value) should be included in the result
```

Structure of sentences

The unit expects that the last sentence in a When clause is the name of service. The rationale is these specifications are defined as a sequence of user interactions with system and the last interaction is the service call. Moreover, we distinguish a kind of constructor methods used to instantiate an object or to create a relationship between two instances, in the Gherkin example *is associated*. For rest of method calls, we assume that the first parameter is the reference of the object caller, eg. in the sentence "...When I call *some-method* with *par1, par2...*" will be translated into *par1.some-method(par2)*.

We have provided 41 utterances for training the sentences in table 4.5 is specified each kind of construction.

Structure	Comments
Given a MN i1 And a MN i2 When i1 $VA1$ and I call [the method] ME with [$PARAMS$] Then EXP	Instance creation.
Given a MN i1 And a MN i2 When i1 $VA2$ i2 and I call [the method] ME with [$PARAMS$] Then EXP	Association creation.
Given a MN i1 And a MN i2 When i1 $VA3$ i2 and I call [the method] ME with [$PARAMS$] Then EXP	Association creation.

Table 4.5: Structure of utterances in validation specs

Abbreviation	Examples
MN	car, bank-account,...
VA1	is created witm | is instantiated with
VA2	is connected to | is associated with
VA3	connects to | associates with
$PARAMS$	i1, i2
EXP	should return i1 | i2 is included

Table 4.6: Data examples

4.2.4 Labeling remarks

To finalize this section we want to mention that we have used less labels that the ones aforementioned in 4.2.1, 4.2.2 and 4.2.3. Actually we have used the same label for attribute-values and for parameters and we called this label sym-value. The reason for that is to minimise the number of labels and thus the odds of classifying wrongly. Moreover we have not created a label for constructors or associative methods but we have used just the label method and we use a list to distinguish this kind of methods. Apart of minimising the set of labels, we will be able to extend in the future this list of words without providing extra utterances to the system and then re-training the unit.

4.2.5 Developments

All the developments are accessible for the reader in the reposity `https://github.com/cmirallesp/MAI-TFM`. The main developments of the NLP unit are: train.py, test.py, data/training_data.json and data/test_data.json. The datasets have been build incrementally, ie. add some simple utterances to the training set, train the classifier and evaluate. The testing script marks visually the errors made by the classifier to facilitate the correction, see in figure 4.2 depicted the output of the test script. The data files contain labeled text used for training and testing, appendix A includes some examples. The web tool **rasa-nlu-trainer**[1] has been used for labeling, see a screenshot in figure 4.3.

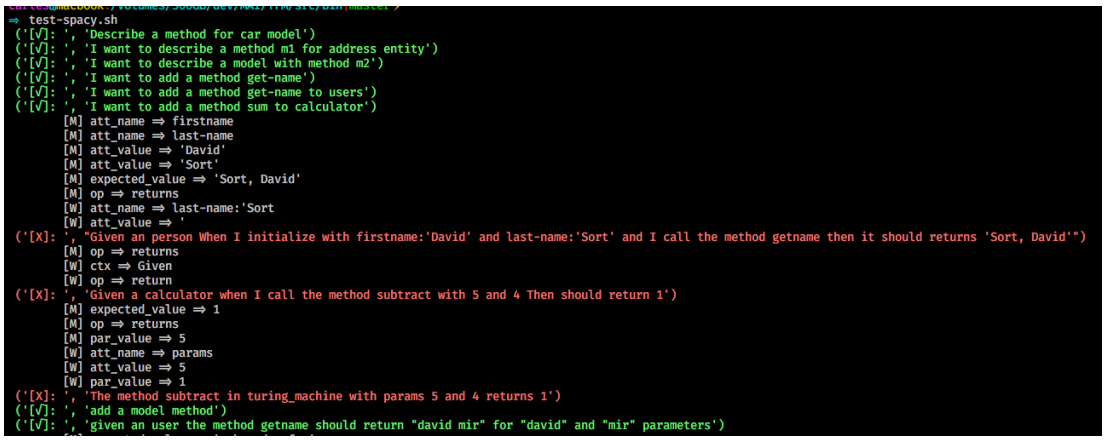


Figure 4.2: Test script output

Usage

```
python nlp_unit/train.py nlp_unit/data/training_data.json <config-file>
python nlp_unit/test.py nlp_unit/data/test_data.json <config-file> <model-folder>
<config-file>: {nlp_unit/config/spacy.json, nlp_unit/config/mitie.json}
<model-folder>: nlp_unit/projects/default/model_YYYYMMDD-HHMMSS
```

[1] `https://rasahq.github.io/rasa-nlu-trainer/`

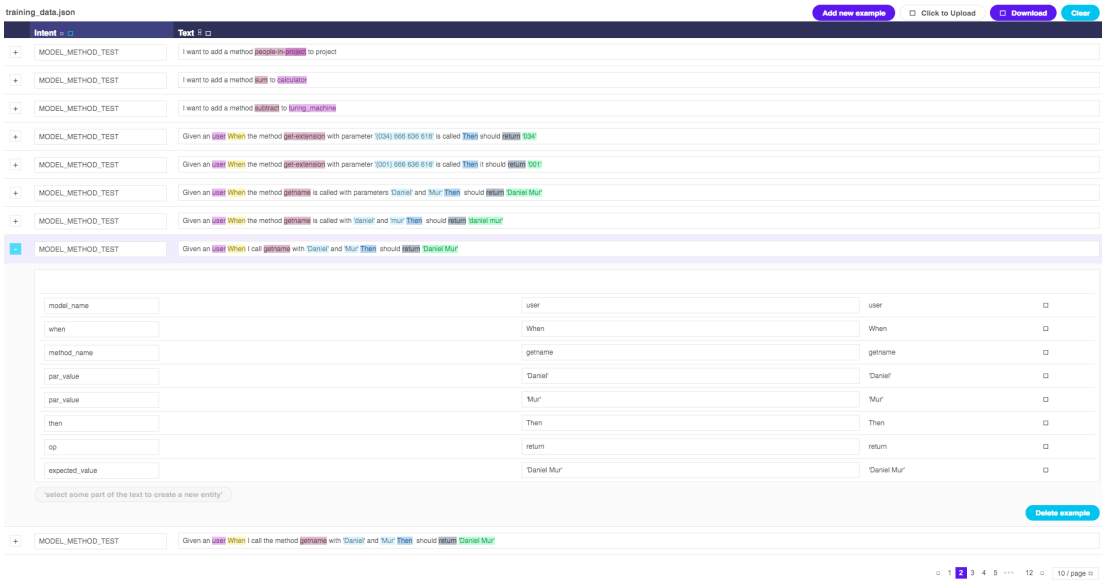


Figure 4.3: Rasa-nlu-trainer labeling tool

4.3 Bot logic unit

The bot logic unit contains the implementation of conversational scripts as well as actions responding to system events. We have used botkit, that let us to implement bots for many instant messages systems such as Slack, Messenger, etc. Botkit is formed by two elements: **SDK** and **studio**.

Botkit SDK is written a Javascript library, thus the code of this unit is implemented in that programming language. It defines a set of events that are triggered in the situations described in table 4.7. In section 4.3.2 is explained how the code of Specbot has been organised to implement the responses to such events.

Event	**Description**
Before script	The code is executed before a conversational script
After script	The code is executed after a conversational script
Before thread	The code is executed before a thread
Validation input	The code is executed after an user input

Table 4.7: Bot Events

On the other hand, Studio is a cloud application in which conversational scripts are defined. A script contains a general topic conversation eg. bot salutation, buy a product, restaurant booking, etc. However the programmer might de-

fine several threads in it containing user-bot interaction with regards to a subtopic or subgoal. An interaction can be a bot output or an user input and we can specify multiple versions of the same sentences to look human-like. One script is executed when some word in the list of activation words is written by the user, however is also possible invoke a script with the SDK.

The script language has also constructions to implement the flow control, ie. **if-then** and **jump** instructions. However we have not used if-then logic in scripts for two main reasons: (i) if-then conditionals are limited to string comparisons, ie. Whether an user answer matches some regular expression, thus other logical predicates as numeric comparisons are not allowed. (ii) To implement the path logic in the scripts makes difficult to debug the bot interaction since the decisions are scattered around the script. Is much preferable to have all the decisions about the steps of the path in one point which is the context management unit. In figure 4.4 is shown a screen shot to design a thread within a script.

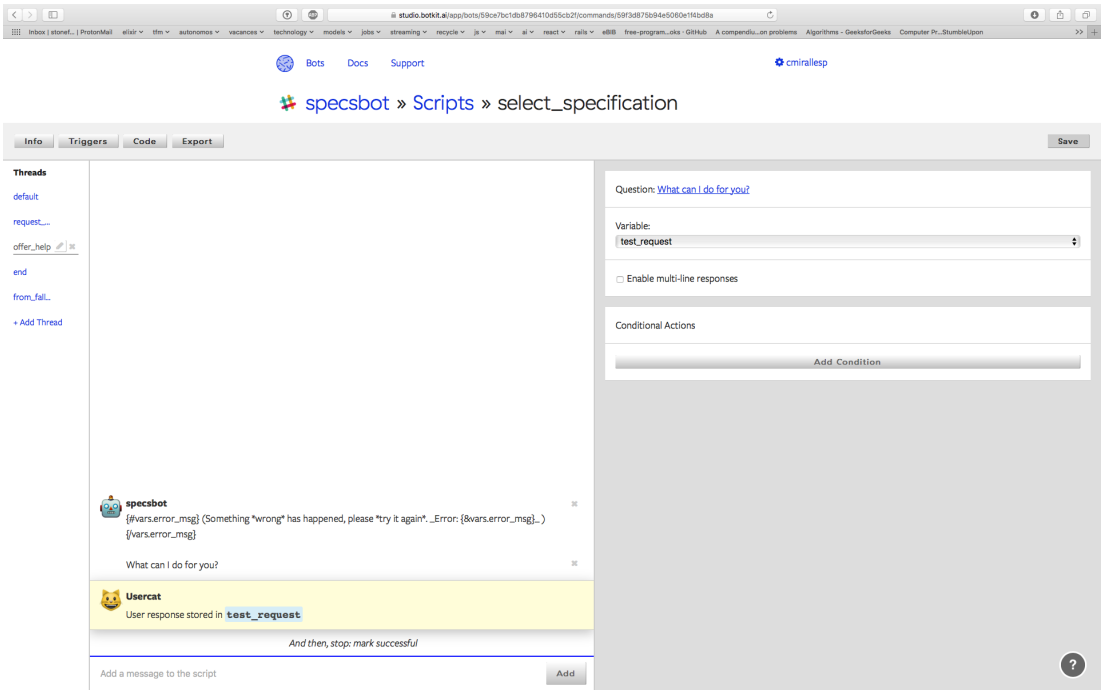


Figure 4.4: Studio dialogue designer

4.3.1 Conversational scripts

We have implemented five conversational scripts **select_specification**, **refine_spec_validation**, **refine_spec_method**, **spec_service** and **fallback**.

Select_specification

Select_specification is the main script and is executed when some salutation word is entered, eg. Hi, hello, etc. It contains four main threads:

- **Default:** the main thread containing the bot salutation.
- **offer_help:** it presents a bot text offering help and waits for user input. Also presents an error message if something wrong has happened in the previous interaction.
- **request_spec:** is showed when the bot is not capable to understand an user requests, and shows a form asking for one specific three kind of requirements.
- **end:** This thread is executed when many errors have happened and says that the bot is not able to continue.

refine_spec_validation

The script refine_spec_validation contains four threads each one to get different parts of validation specification:

- **ask_model_name:** shows a message asking for the model name.
- **ask_attributes:** shows a message asking for a list of attributes required to have a valid model.
- **ask_values:** ask one value for each attribute introduced in ask_attributes.
- **show_spec:** shows the executable specification.

refine_spec_method

The script refine_spec_method nine different threads, each one with a different subgoal:

- **ask_model_name:** shows a message asking for the model name.
- **ask_attributes:** shows a message asking for a list of attributes required to have a valid model.
- **ask_values:** asks one value for each attribute introduced in ask_attributes.
- **ask_method_name:** asks for the name of the method being specified.
- **ask_params:** asks for the method parameters.
- **ask_method_value:** asks for the returned value to be checked.
- **show_spec:** shows the executable specification.

spec_service

The script spec_service has two threads:

- **ask_requirement** which just waits for a requirement.
- **show_spec** which shows the executable specification.

fallback

The fallback script does not contain any text, is just an artificial script that implements error handling in its class counterpart.

4.3.2 Bot code

The classes described in this section contain multiple handlers associated to any kind of event described in 4.7. The base code is in folder **bot/es6/skills** and contains six javascript classes.

- **SkillsBae** implements the code shared by the rest of the subclasses. It responds to before-script event which calls the abstract method create_fsm to instantiate a class from the CMU used to know the next script/thread in the dialogue. In the after-script event saves the current state in a stack that might be recovered if the user comes back to the script.

- **SelectSpeficiation** rewrites create_fsm method which instantiates the SelecSpecificationFsm.

- **RefineSpecOneModel** implements the event handlers for each user input, ie. model name, attribute names, attribute values, method name, parameters and expected value.

- **RefineSpecValidation** rewrites create_fsm method and instantiate RefineSpecValidationFsm.

- **RefineSpecMethod** rewrites create_fsm method and instantiate RefineSpecMethodFsm.

- **Fallback** restores the context from the failing scripts and returns the control to it.

In figure 4.5 is depicted the UML diagram.

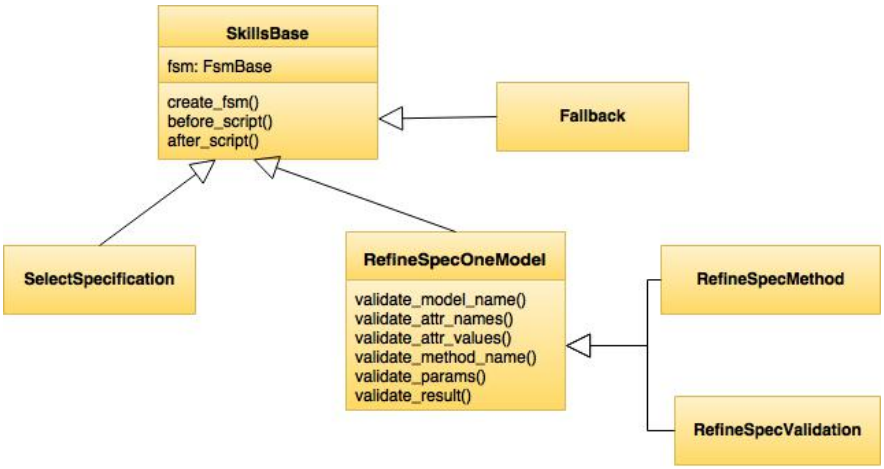


Figure 4.5: Bot logic unit classes

4.4 Context managament unit

Context management unit (CMU) implements user-bot conversation flows as well as the classes responsible for generating executable specifications. The flow has been implemented by means of finite state machines (FSM) in folder **bot/es6/lib/fsm**. The classes that generate executable specification are in folder **bot/es6/lib/specs**. In the following sections are described the three FSM and the generation of executable specifications.

4.4.1 SelectSpecificationFsm

The SelectSpeciFsm receives an user utterance in natural language. It figures out the text intention and drives the user to choose one of the three kind of specifications when the given text is unknown by the NLP unit. In figure 4.6 shows how the FSM proceeds once receives an answer from the NLP unit which includes a classification, a confidence level and the list of identified entities. Therefore if the level of confidence is high enough we pass the control to the next script, ie. refine validation, refine method or refine service. If the confidence is low it is assumed that the user intention is another one and then goes to a state in which the bot should ask for one valid type. When the user has selected one of the three types, the FSM loads the next script ie. refine validation, method or service.

4.4.2 RefineSpecValidationFsm

This FSM has four states, each one meant to get a piece of required information from the user. The first state asks for the name of the model, then asks for the list of attributes that are required to validate the model. After that the next state requests the values of each attribute and eventually shows the executable specification. Figure 4.7a) depicts the complete state machine, the transitions are triggered when the user provides the required input.

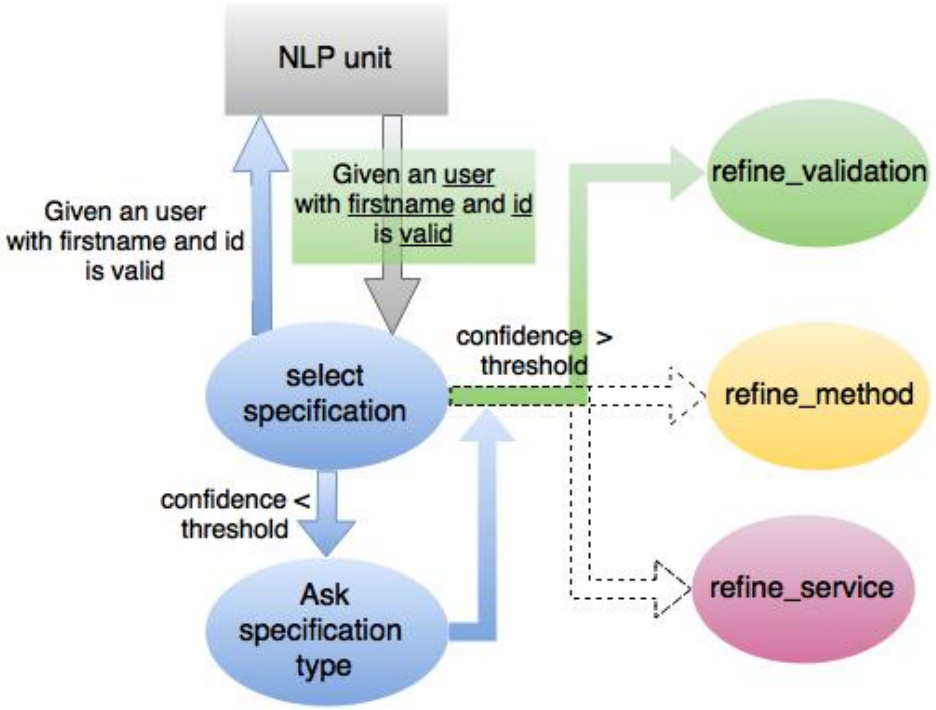


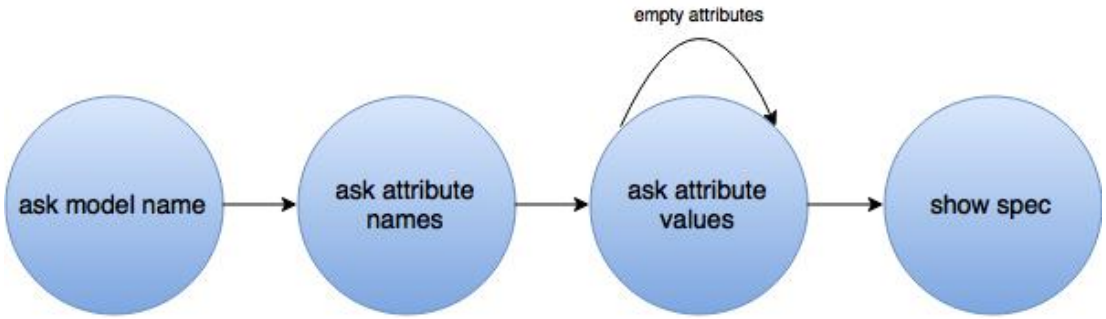
Figure 4.6: Select specification

4.4.3 RefineSpecMethodFsm

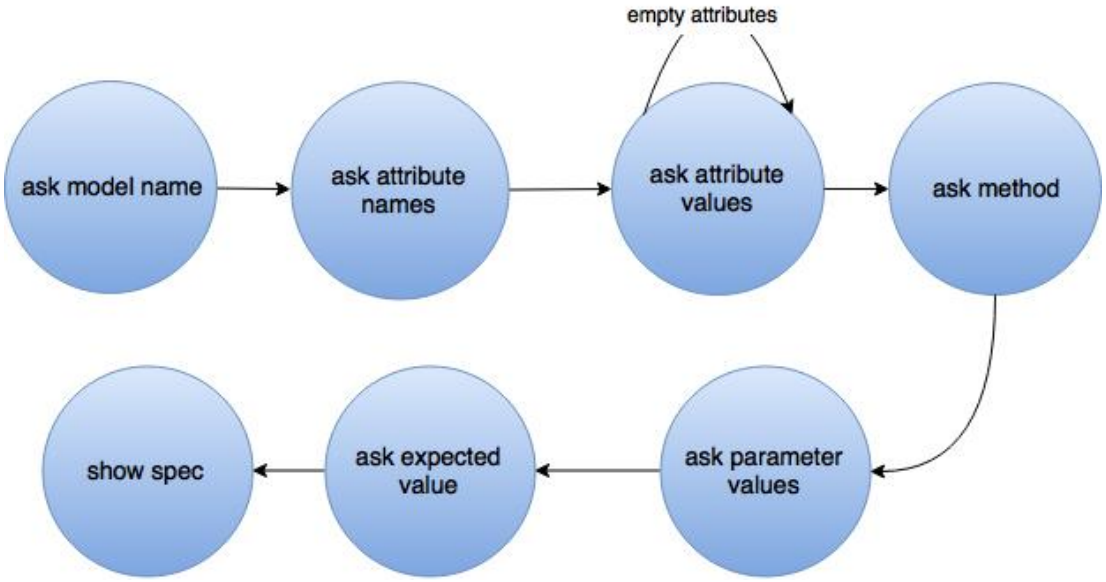
The FSM navigates through eight states to get all the information required to specify the method specification. The three first states and the last one are the same as in refine validation FSM. The state ask method gets the name of the method, which is followed by the list of parameters and the result of the method. Figure 4.7b depicts the complete navigation.

4.4.4 Generating executable specifications

The CMU is also responsible for translating requirements into executable specifications. This functionality might have been located in the bot logic, however since the CMU has got all the required information is more convenience to put here such classes. In figure 4.8 is depicted the classes implemented: Spec is the main class and which is responsible for generating the executable specification. It has one or many instances and one or many expectations. An instance contains a list of attributes to save an internal state. An expectation models a method expectation and has a list of parameters as well as an expected value. A symbol



(a) FSM model validation



(b) FSM method specification

might contain a parameter, an attribute or an expected value.

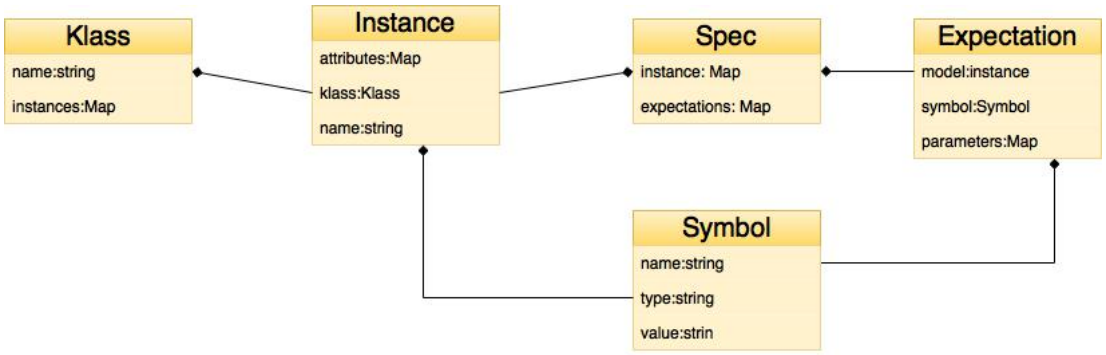


Figure 4.8: Executable specification classes

Chapter 5

Evaluation

5.1 Introduction

In this section the evaluation of the NLP unit is introduced. Rasa/NLU has two libraries to calculate intent classification and entity recognition, ie. MITIE and SpaCY . However we only show results based on Spacy recogniser that uses conditional random fields (CRF). The reason for that is because the time required to train MITIE with a SVM became unacceptable with the current number of examples in the training set. While SpaCY train in less than a minute MITIE takes hours, for that we reject this library to be used in this project.

5.1.1 Test evaluation

We have built a test set with 32 utterances to evaluate both intent classification and entity recognition task. In terms of intent classification, Spacy uses a SVM with a linear classifier, and obtained perfect results with a 100% of accuracy. For entity recognition it uses CRF got a global accuracy of 95%. If we pay attention in each label in table 5.1, is remarkable that the label service_name makes a 25% of false negatives, ie. missed values. The second worst result is the label instance_name with a 20% of missing labels. The rest of labels got in general good results in both errors. We analyzed the output of the test script and we detected that the classifier confuses (i) service_name with method and (ii) instance_name with W: sym_value. As a conclusion, we can say that the

results are good in general, however we should consider to include few additional examples to diminish these detected errors.

LABEL	TOT	OK	WR	% WR	MI	% MI	F1
model_name	37	36	2	0.05	1	0.03	1.13
instance_name	25	20	4	0.17	5	0.2	4.44
sym_value	50	47	2	0.04	3	0.06	2.4
service_name	8	6	0	0.0	2	0.25	0
expected_value	16	16	0	0.0	0	0.0	0
method	38	36	4	0.1	2	0.05	2.67
att_name	32	32	1	0.03	0	0.0	0
op	23	23	0	0.0	0	0.0	0
TOTAL	280	266	13	0.0464	13	0.0464	13

Table 5.1: Results with test dataset

5.1.2 Edge cases evaluation

In this section we want to evaluate the trained model with sentences with more complexity than the described in section 4.2.

The first edge case evaluated was a validation test receiving more than four attributes that as we said previously is the maximum number in the training set, eg *Given model When is created with* $at_1 : val_1$, $at_2 : val_2$, $at_3 : val_3$, $at_4 : val_4$, $at_5 : val_5$, $at_6 : val_6$ *Then we expect to be valid.* The result was excellent with the two test exanples, one with 5 attributes and another one with 7.

The second edge case was to check the limits in service specification. To do that we provide an example with more of two sentences in the clause *When* and we mess active and passive voice in which the order of the parameters is inverted, ie. *Given an animal a1 and an animal a2 and an animal an3 When a1 is linked to a2 and a1 links to a3 and I call m2 with a1, 2 and 4 then I expect to get 'a2'"*. We observed in table 5.2 new errors that the classifier did not get before which tell us that now we must provide much more examples to get better generalization.

Text	**Expected**	**Obtained**
a1 (When)	sym_value	instance_name
a2 (linked to)	sym_value	model_name
a1 (with)	sym_value	att_name

Text	Expected	Obtained
m2	service_name	expected_value

Table 5.2: Errors obtained in the second edge case

5.1.3 Final comments

In this section we want to mention an additional approach to be implemented in the NLP unit for getting less errors and obtain a more robust solution. As we have seen in 4.2 we can express textual requirements using a limited set of sentence structures. So that we might build a small grammar or a chunker with those structures. Then, since the recogniser returns an ordered list of pairs (text,entity), if we also train the ER to learn the constants Given/When/Then/And we will obtain a chunk as: [(**Given**,given), (Person,model_name), (p1,instance), (**And**, and), (Car, model_entity), (c1, instance), ..., (**When**,when), (p1,instance), (associates,method), (c1,sym_value)..., **Then**....] With that we will be able to check if each clause is accepted by the grammar. We propose this approach as a future work.

Chapter 6

Planning

In this chapter is explained the temporal planning followed for the execution of this thesis. The thesis has 18 ECTS[1] and a credit corresponds to 30 hours of workload, thus the total expected time is 540 hours,ie. 67 days. In figure 6.1 is depicted gantt chart in is identified four milestones: (i) research and proposal, (ii) bot implementation, (iii) NLP Unit and (iv) documentation.

In the research and proposal phase a review of the bibliography and the current state of the art was done. It also includes a proposal in form of internal document to describe the rest of the project and a planning. The duration of this milestone was eleven days.

The bot implementation phase was done in parallel with the construction of the NLP unit and lasted sixty days. The breakdown is composed of four tasks:

1. Installation and setup systems: this technical task consisted of downloading, installing and testing all the libraries required for this project. It also included an initial installation in a server to have the services available in the cloud. The estimated time slot is one day.

2. Bot logic task: in which was implemented each conversational scripts and code to respond the system events.

3. Context management: to implement the finite state machines to control the

[1]European Credit Transfer and Accumulation System

bot flow.

4. RSpec generator: was the implementation of the classes to obtain executable specifications in RSpec.

The NLP unit phase consisted of the construction of the dataset. A total of 140 textual requirements expressed in Gherkin and 1207 labels. The time slot allocated for this phase was fifty days.

Finally, the last phase was the documentation writing and the preparation of the oral defense that lasted a total of thirty days.

The actual time dedicated was 75 days which supposed an $\approx 11\%$ of deviation with respect the formal time. The main difficulties responsible for the divergence were: (i) the lack of experience building chatbots in general and with the selected technologies in particular. (ii) testing bot interactions required check manually each branch in a path. Moreover, the use of the API of Slack has a technical constraint in which each call lasts 1.5 second. (iii) Built and label textual data is a time-consuming task requires to learn the tag language of the library, to select utterances and label them. The major part of this part was done manually with a plain editor since I did not know the existence of the tool.

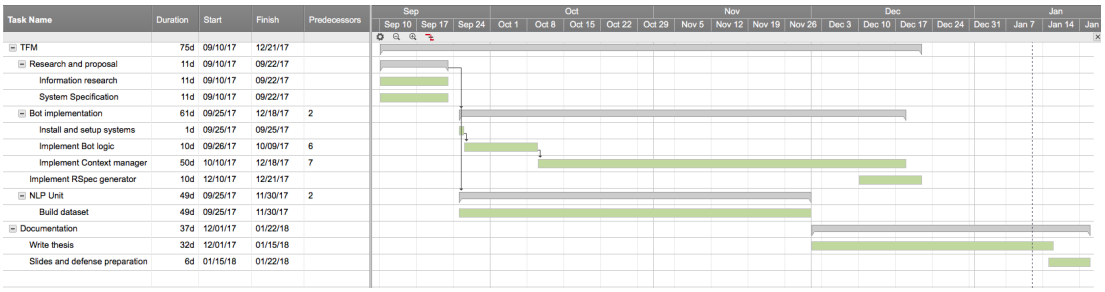


Figure 6.1: Planning

Chapter 7

Conclusion

The goal of this thesis was to build a chatterbot to capture textual requirements and generate automatically executable specifications. I approached such transformation identifying meaningful parts in a requirement that might be solve with any entity recognition library that accepts custom entities. The lack of previous works for the translating task meant the absence of ground data to train and evaluate any algorithm, for that reason I had to built a labeled dataset. Label textual data is a high time consuming task that requires find data, identify the entities and eventually tagging. Moreover has to be done in parallel with the process of training and evaluating the recogniser to debug which errors appear in each iteration and thus, add new examples to improve the result. First, I defined in 4.2 a limited set of sentences structures to be taken into account during training the tool. After having a minimum dataset I conducted experimentation with the test set to evaluate the proposed approach and analyse its limits. I observed numerically in 5.1.1 that the approach obtained good results with a 95% of accuracy. However in 5.1.2 I observed that when I increment the number of sentences, although they followed the same structure aforementioned, appeared new errors. I pointed out that the NLP unit require more examples labeled in the training set. I also proposed as future work an improvement in 5.1.3 to make more robust the tool based in syntax parsing with chunkers.

With respect to the chatterbot I got some lessons learnt from a real experience with the tool chosen to implement it. The current tools in the market are

based on imperative programming languages. It means that I could not express the bot goals declarative as in BDI architectures that facilitate the agent cycle . Instead, I proposed several deterministic state machines to define a flow in 4.4, however I had to code and test them that requires much more effort than declarative approaches. During the testing time I realised that each message was processed slow (1.5sec) because of technical constraints of the Slack API. In a future, it would be preferable select a tool if exists that let me work with a simulator.

7.1 Future Work

As a future work I propose several improvements. First, to implement the idea of using a grammar introduced in 5.1.3 and evaluate the result of the classifier with it to detect and even correct mistakes. I believe that we might detect many syntax inconsistencies from the classifier and even automatically fix some mistakes by means of rewriting rules.

During the execution of this thesis I found out two ontologies: DBPedia [1] or OpenCyc [4]. They might be applied to propose attribute values in model validations from the attribute names which to simplify this kind of requirements. Other applications might be found after exploring these sources of knowledge.

Another proposal is a way for improving dynamically the training set from the feedback of the user. That is, each time that the bot shows an executable specification asks for the feedback, figuring out which parts was mistaken, and adding the error to the training set that should be processed from time to time.

The identification of sentences such as *Given two users u1 and u2* might also interesting to simplify the textual requirements.

Appendix A

In this appendix we include some examples of labeled data used to train our model.

```
## intent:MODEL_METHOD_TEST
- add a model's method
- describe a model's method
- I want to describe a method of a model
- I want to add [m2](method_name) to a model
- [Given](ctx) an [user](model_name) When the method [getname](method_name)
is called with parameters ["Daniel"](par_value) and ["Mur"](par_value)
Then  should [return](op) ["Daniel Mur"](expected_value)
- [Given](ctx) a [calculator](model_name) When the method [sum](method_name)
with parameters [3](par_value) and [2](par_value)
Then is should [return](op) [5](expected_value)
- [Given](ctx) a [list](model_name) When I call the method [add](method_name)
with parameters [5](par_value) and [7](par_value)
Then should [return](op) [\[5,7\]](expected_value)

## intent:MODEL_VALIDATION
- I want to describe a validation
- add a validation
- describe a validation
- I would like to add a [user](model_name) validation
- I want to describe a validation for [user](model_name) model
```

```
- validation for [houses](model_name)
- describe a validation for [projects](model_name) model
- [given](ctx) a [project](model_name) When is created with attributes
[name](att_name), [num-of-tasks](att_name) and [num-of-people](att_name)
then it should [be](op) [valid](method_name)
- [given](ctx) a [project](model_name) When a new instance
with [name](att_name) [prj1](att_value), [num-of-tasks](att_name)
[3](att_value) and [num-of-people](att_name) [5](att_value) is created
Then should [be](op) [valid](method_name)
- [given](ctx) a new instance of [project](model_name) When is initialized with
[name](att_name) [prj1](att_value), [num-of-tasks](att_name) [3](att_value)
and [num-of-people](att_name) [5](att_value)
Then is expected to [be](op) [valid](method_name)
- [given](ctx) a [bank account](model_name) When is created with
[holder](att_name), [expiration date](att_name) and [cvv](att_name)
then I expect to [be](op) [valid](method_name)
- [given](ctx) a [bank_account](model_name) When is created with
[holder](att_name), [expiration_date](att_name) and [cvv](att_name)
then should [be](op) [valid](method_name)
```

Bibliography

[1] Dbpedia. http://wiki.dbpedia.org.

[2] Eliza implementations. https://en.wikipedia.org/wiki/ELIZA#Partial_list_of_implementations.

[3] Gherkin wiki. http://github.com/cucumber/cucumber/wiki/Gherkin.

[4] opencyc. http://opencyc.org.

[5] Rasa pipeline. https://nlu.rasa.ai/pipeline.html.

[6] Software testing. https://en.wikipedia.org/wiki/Software_testing.

[7] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[8] X. Carreras, L. Màrquez, and L. Padró. A simple named entity extractor using adaboost. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 152–155. Association for Computational Linguistics, 2003.

[9] D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. The rspec book: Behaviour driven development with rspec. *Cucumber, and Friends, Pragmatic Bookshelf*, 2010.

[10] M. Chernosky. How to write better unit tests for embedded software with tdd, March 2016.

[11] S. H. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceed-*

ings of the international conference on education and information systems: technologies and applications EISTA, volume 3, 2003.

[12] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[14] D. S. Janzen and H. Saiedian. Test-driven learning: intrinsic integration of testing into the cs/se curriculum. In *ACM SIGCSE Bulletin*, volume 38, pages 254–258. ACM, 2006.

[15] H. S. Nwana. Software agents: An overview. *The knowledge engineering review*, 11(3):205–244, 1996.

[16] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[17] I. C. Society, P. Bourque, and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[18] J. Weizenbaum. Eliza a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[19] D. Zhang and W. S. Lee. Question classification using support vector machines. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 26–32. ACM, 2003.